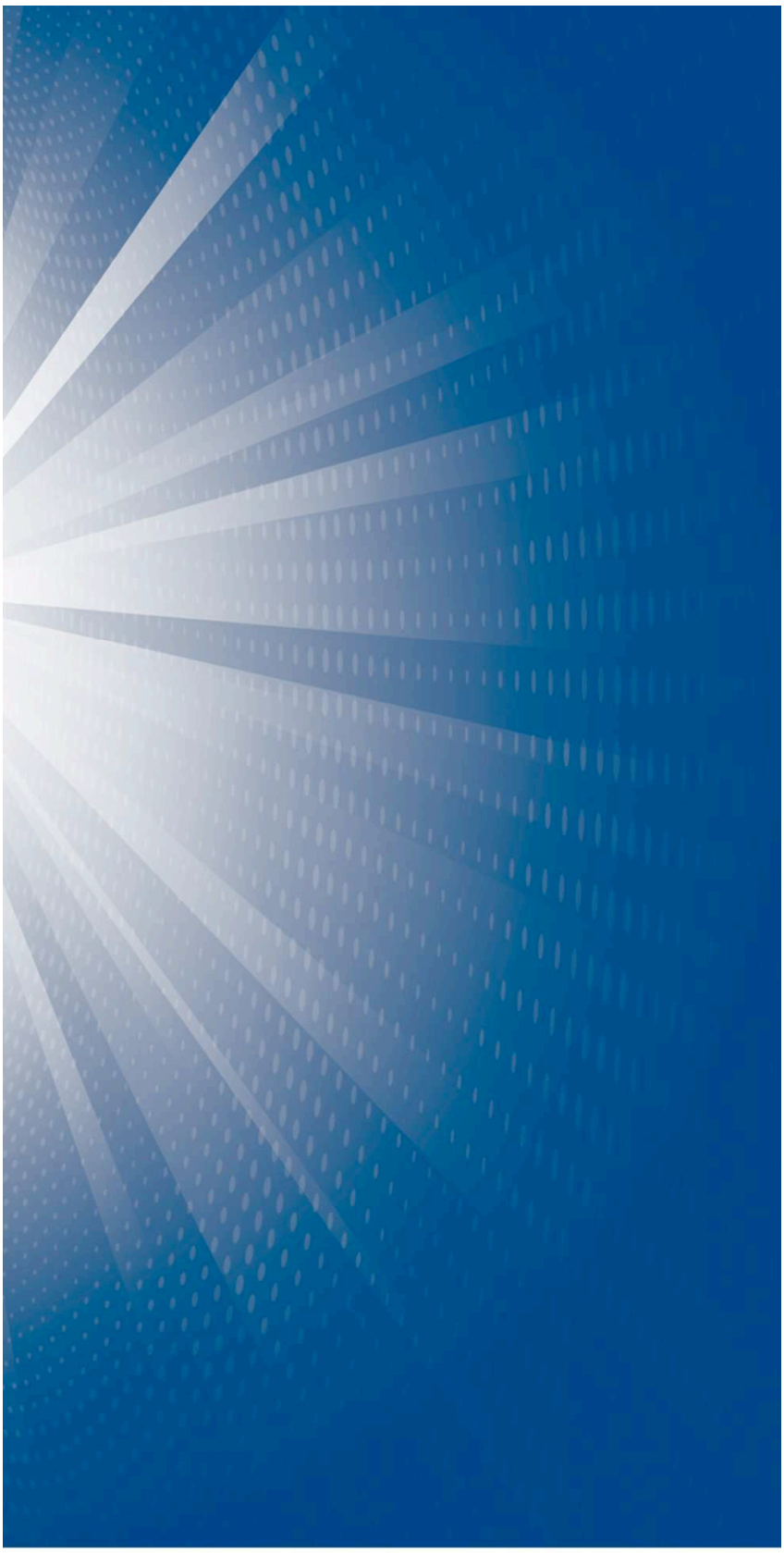
EXHIBIT 53



Source Code Analysis of gstumbler

Prepared for Google and Perkins Coie
Prepared by STROZ FRIEDBERG
June 3, 2010

STROZ FRIEDBERG

Table of Contents

I. Introduction

1. Stroz Friedberg, LLC ("Stroz Friedberg") is a consulting and technical services firm that specializes in digital forensics, data breach and cyber-crime response, on-line and traditional investigations, and electronic discovery. The firm was founded in February 2000 by Edward M. Stroz. For ten years, Mr. Stroz has been a leader in the computer security and digital forensics field, and has pioneered the use of a blend of behavioral science and digital forensics in addressing the insider threat. Before founding what was then Stroz Associates, Mr. Stroz founded and then ran the Computer Crimes Unit of the F.B.I.'s New York office during his sixteen year career with the Bureau. Eric Friedberg, Mr. Stroz's Co-President at Stroz Friedberg, hails from the U.S. Attorney's Office in the Eastern District of New York, where he was the lead cyber-crime prosecutor and the Chief of the Narcotics Unit during his eleven year tenure as an Assistant United States Attorney there. Mr. Friedberg is an expert in cybercrime response, computer forensic investigations, and electronic discovery. Mssrs. Stroz and Friedberg, together with the firm's Executive Management, manage the firm's operations. Stroz Friedberg's principal offices are in New York (HQ), Los Angeles, Washington, D.C., London, Dallas, Minneapolis, San Francisco, and Boston. The firm has handled many significant, high-profile digital forensics matters, including a number of source code analyses in the civil, regulatory, and criminal arenas. Mr. Friedberg led the team that conducted the source code analysis in this case.

2. Stroz Friedberg was retained by Perkins Coie, on behalf of Google, to evaluate the source code of an executable deployed on the vehicles otherwise collecting data for Google's Street View service offerings. Specifically, we were asked to provide a third-party assessment of the functionality of the source code for a Google project named "gstumbler" and its main binary executable, "gslite," with particular focus on the elements of wireless network traffic that the code captured, analyzed, parsed, and/or wrote to disk. Stroz Friedberg has no stake in the outcome of this matter and has been asked by Google and Perkins Coie to render a neutral, technical opinion regarding the functionality of gstumbler. Stroz Friedberg is being compensated on a time and materials basis. The project team consisted of three primary examiners/code reviewers and two engagement managers, and our report was internally peer-reviewed by others in the firm.

3. Between May 20 and May 26, 2010, Stroz Friedberg received the gslite source code from Google. The gslite source code is comprised of approximately thirty-two source code files, along with twelve additional files including configuration files, shell scripts, source code repository changelog information, binary executables, and kernel modules. A full inventory of the reviewed source code files and shell scripts is provided in Appendix A. It is our understanding that the provided source code and accompanying shell scripts represent the most current version of the gstumbler application deployed as of May 6, 2010, on vehicles otherwise capturing data for Google Street View. Our findings regarding the application's functionality, based upon our review of the source code, are set forth below: first, in the Executive Summary, and then more specifically in the Overview of Findings and the body of this report.

A. Executive Summary

4. The executable program, gslite, works in conjunction with an open source network and packet sniffing program called Kismet, which detects and captures wireless network traffic. The program facilitates the mapping of wireless networks. It does so by parsing and storing to a hard drive identifying information about these wireless networks – including but not limited to their component devices' numeric addresses, known as MAC addresses, and the wireless network routers' manufacturer-given or user-given names, known as "service set identifiers," or "SSIDs." The "parsing" involves separating these identifiers into discrete fields. Gslite then associates these identifiers with GPS information that the program obtains from a GPS unit operating in the Google Street View vehicle. Gslite captures and stores to a hard drive the header information for both encrypted and unencrypted wireless networks.

5. While gslite parses the header information from all wireless networks, it does not attempt to parse the body of any wireless data packets. The body of wireless data packets is where user-created content, such as e-mails or file transfers, or evidence of user activity, such as Internet browsing, may be found. While running in memory, gslite permanently drops the bodies of all data traffic transmitted over encrypted wireless networks. The gslite program does write to a hard drive the bodies of wireless data packets from unencrypted networks. However, it does not attempt to analyze or parse that data.[1]

B. Basic Technical Descriptions and Definitions

6. To understand the functionality of the gslite source code, and to understand the Overview of Findings set forth below in Section 1(C), it is important to understand the basic technical concepts critical to the architecture of wireless 802.11 networks and the transmission of data over such wireless networks.

7. Data is transmitted over the Internet via packet switching technology. Briefly, a communication transmitted via the Internet is broken up into "packets" at the point of origination, and the packets of data are routed from the originating device to various other computer devices on the Internet until they reach their final destination. Each packet is comprised of a packet header which contains network administrative information and the addressing information (or "envelope" information) necessary to transmit the data packet from one device to another along the path to its final destination. Each packet also contains a "payload" which is a fragment of the "content" of the communication or data transmission sent and received over the Internet; payload information can include, for example, fragments of requests for URLs, files transferred across the Internet, email bodies, and instant messages, among other things. The packets associated with a particular data transmission may travel over different routes across the Internet to reach their final destination; once they reach the destination device, the packets are reassembled to create the entire transmission.

8. A router is a device on a network that receives a data packet and transmits it to the next router or device on the network. A MAC address is a unique number assigned to a piece of networking hardware, such as a router, by that hardware's manufacturer. Each device and router on a wireless network has a MAC address uniquely identifying that machine.

9. Packets are encapsulated into larger data packages called frames for routing over various network types. Multiple specifications for the transmission of packets using frames have been promulgated by the Institute of Electrical and Electronics Engineers. This report focuses on

[1] From an analysis of the source code alone, we cannot ascertain the extent to which gslite captures of unencrypted wireless data would be fragmented or complete. Given the factors that the Google Street View vehicles can be moving or stationary and, as discussed below, the Kismet device is set to hop rapidly between wireless channels, the numerous wireless data packets that constitute any single user communication may or may not be captured by Kismet.

data transmitted over wireless networks pursuant to the 802.11 protocols, the specifications for which provide the international standard for the transmission of data over wireless networks operating in the 2.4, 3.6, and 5 GHz frequency radio bands.

10. There are three primary types of 802.11 frames, which contain information necessary to transmit data packets from one device to another over wireless networks. The three types of 802.11 frames are Control frames, Management frames, and Data frames, each of which is described below:

a. *Control Frames* control access to particular types of networks and facilitate exchanges of Data frames between wireless links. Control frames send the Request to Send (RTS) and Clear to Send (CTS) messages necessary to establish a connection between two links on a network prior to transmitting a data packet (sometimes referred to as a "two-way handshake"). Control frames also transmit the Acknowledgement (ACK) information once a Data frame is received by a link. A diagram of a generic Control frame is provided in Appendix B.1.

b. *Management Frames* contain information necessary to manage a data transmission over the network. Management frames contain, for example, authentication information, information necessary to allocate resources to a transmission, data transmission rates, SSIDs (i.e., network names), information necessary to terminate a connection, and periodic beacon signals. These properties are stored, in part, as Information Elements, that is, id-value pairs in the payload of Management frames. A diagram of a generic Management frame is provided in Appendix B.2.

c. *Data Frames* serve the function of encapsulating and transmitting packets of data over wireless networks. Generally, the body of each Data frame contains the "content" data of the encapsulated packet transmitted over the Internet, including such user-created data as email header information and bodies, URL requests, file transfers, instant messages, or any other communication over the Internet, as well as the addressing information for such transmissions. A diagram of a generic Data frame is provided in Appendix B.3.

d. Each of these frame types have numerous subtypes, which determine, among other things, the fields present in the 802.11 frame. A frame's type and subtype information is stored in the *Frame Control* header field of the 802.11 frame, which is discussed in more detail below.

11. At a high level, an 802.11 frame can be considered to have two distinct sections: the header data and the body data. The header data is comprised of the Frame Control, duration or id, MAC addresses, sequence control number, and quality of service, or QoS, control information. The body data is comprised of the frame body component of an 802.11 frame, to the extent the frame's type and subtype calls for this field. As noted, the body of a Data frame may contain packet content data.

12. A diagram of a generic 802.11 frame showing its various components is below:

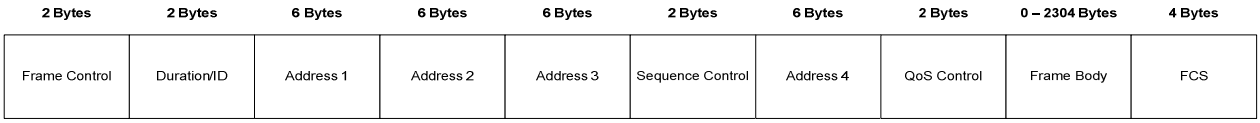


Figure 1. Generic 802.11 Frame Format.

The Frame Control, Duration/Id, Address, Sequence Control, and QoS control fields are considered the 802.11 *frame header*, while the frame body contains the payload data previously discussed. The FCS field contains checksum information used to confirm that the wireless frame was accurately received.

13. Every 802.11 frame contains a 16 bit Frame Control field that contains information regarding the status of the frame and the wireless transmitter of the frame. Specifically, the Frame Control field contains the following properties: Protocol Version; Type; Subtype; To DS; From DS; More Fragments; Retry; Power Management; More Data; Protected Frame; and Order. The Type field is a two bit field that will be 00, 01, or 10 to indicate if a frame is a Management, Control, or Data frame respectively, and the Subtype is a four bit field used to specify the frame's subtype. The To DS and From DS fields are single bit values that specify the routing of the 802.11 frame across the wireless network.

14. The Protected Frame bit in the Frame Control field is also known as the frame's "encryption flag." The Protected Frame field is a single bit which identifies whether the wireless network's transmissions are encrypted; it has no relation to the payload within any Data frame or whether that encapsulated packet transmission is itself independently encrypted. For example, if a fragment of a secure, encrypted HTTP session (HTTPS) were encapsulated in the payload of a Data frame on an unencrypted wireless network, the Data frame's encryption flag would still be set to "0", i.e. "false", indicating that the frame is unencrypted. The 802.11w-2009 amendment to the 802.11 specification, which was approved on September 11, 2009, provides a mechanism to also encrypt unicast Robust Management frames, which will result in the Protected Frame field being set to "1", i.e. "true."

15. Each 802.11 frame type contains at least one MAC address associated with the wireless local area network (LAN). 802.11 frames can contain up to four such MAC addresses associated with a particular wireless LAN.

16. Each wireless network has a public name, known as the SSID. The SSID name may be set by the owner of the wireless network. The SSID can be publicly broadcast to all wireless devices within its range. The broadcast feature also can be disabled so that the SSID for a particular wireless network is not readily visible to devices seeking wireless networks even though the SSID is still ascertainable from the transmitted packets.

17. The 802.11 wireless specifications divide each of the frequency bands into *channels*, analogous to TV channels. The division is regulated by individual countries, resulting in different locales having different numbers of permitted channels in each band. For example, in European countries, the frequency bands are regulated such that transmission is permitted across thirteen overlapping channels between 2.4 and 2.4835 GHz, each of which is 5 MHz apart and 22 MHz in width. A particular communication is transmitted over only one channel; thus, to the extent a packet sniffer is set to "hop" through channels—similar to changing a radio or TV channel—it may only collect fragments of a particular communication.

C. Overview of Findings

18. Using the more technical terminology in the above section, we expand on our high-level findings.

19. As set forth above, the executable program, gslite, is an 802.11 wireless frame parsing and collection tool that associates GPS coordinates with wireless network frames. While running in memory, the program parses frame header information, such as frame type, MAC addresses, and other network administrative data from each of the captured frames. The parsing separates the information into discrete fields for easier analysis. In addition, per-packet information regarding the wireless transmission's strength and quality is captured and associated with each frame. All available MAC addresses contained in a frame are also parsed. All of this parsed header information is written to disk for frames transmitted over both encrypted and unencrypted wireless networks.

20. The gslite program discards the frame bodies of 802.11 Data frames sent over encrypted wireless networks. The program inspects the encryption flag contained in each frame header to determine whether the frame is encrypted, i.e., whether it is being transmitted over an encrypted wireless network. If the encryption flag identifies the wireless frame as encrypted, the payload of the frame is cleared from memory and permanently discarded. If the frame's encryption flag identifies the frame as not encrypted, the payload-–which exists in memory in a non-structured bit stream of ones and zeros--is written to disk in a serialized format, as further described below.

21. The gslite program parses Management frame bodies and stores the parsed data as "Information Elements." The gslite program also parses Control frames' subtype information before writing it to disk. By contrast, gslite does not parse Data frames' bodies, which may contain user-created content. Rather, unencrypted Data frames' bodies pass through memory unparsed and are written to disk in their unparsed format. (Again, encrypted frame bodies are dropped entirely.)

22. As set forth above, the gslite source code includes logic that examines wireless frames' type and encryption status, and determines whether to discard them in whole or in part. The default behavior of gslite is to record all wireless frame data, with the exception of the bodies of encrypted 802.11 Data frames. The gstumbler application is configurable through the use of command line arguments that make it possible to specify, at the time the program is run, what types of wireless frames to record. Based on our review of the provided configuration files and shell scripts used to launch gslite, prior to May 6, 2010, the gstumbler application used the default configurations described above, which is to say that all wireless frame data was recorded except for the bodies of 802.11 Data frames from encrypted networks.[2]

II. Overview and History of gstumbler, gslite, and Kismet

23. The source code reviewed is from a project referred to at Google as "gstumbler." According to internal Google documentation, gstumbler was first created and used in 2006. At that time, the program executable was itself also named "gstumbler," but at some point in or after late 2006, the executable deployed to vehicles otherwise capturing data for Google's Street View services was revised and renamed "gslite." The gslite program is the focus of this source code review. In this report, "gslite" refers to the specific executable program for which Stroz Friedberg reviewed the source code; and "gstumbler" refers to the overall application, including the configuration files and shell scripts that the Google Wifi project has used to detect and collect wireless network data.

24. The gslite source code is written in C++. C++ is an object oriented programming language, where objects are defined as data structures comprised of properties and methods, i.e. values and functions. An "object" refers to an instance of a data structure in memory. The gslite program makes use of object oriented programming to represent 802.11 frames in memory, parsing the raw frame data and storing its structural elements in a Dot11Frame object as defined in the source code file packet.proto. The Dot11Frame object is defined using a framework called Protocol Buffers, which was developed at Google to provide a means for writing complex data structures to disk. Protocol Buffers are discussed more fully in Appendix C.

25. The gslite program parses some, though not all, information from 802.11 wireless frames read in from a source of wireless frames. It simultaneously receives geolocation coordinates from a GPS system and then associates each wireless frame with the time and approximate location in which it was received. The gslite program works in concert with a second program, Kismet, which must run simultaneously. Kismet controls one or more wireless cards on a Google vehicle

[2] It is our understanding that on May 6, 2010, in response to regulatory attention, the gstumbler shell script was revised to disable *all* Data frame capture. We have inspected that revised shell script and have confirmed that revision.

and provides gslite with the stream of detected wireless frames. The relationship between gslite and Kismet is depicted in Figure 2.

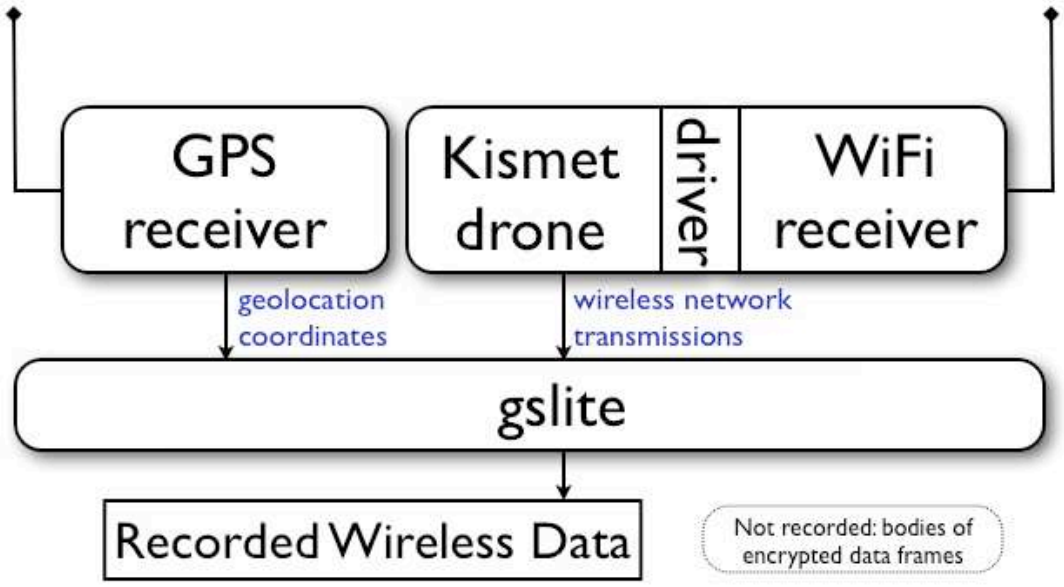


Figure 2. Inputs to gslite.

26. Kismet is a freely available, open-source application for wireless network detection and packet sniffing. Kismet captures wireless frames using wireless network interface cards set to monitoring mode. The use of monitoring mode means that Kismet directs the wireless hardware to listen for and process all wireless traffic regardless of its intended destination. Kismet captures wireless frames passively, meaning that that Kismet receives such transmissions without actively transmitting to nearby wireless networks. Kismet only detects packets passively. Through the use of passive packet sniffing, Kismet can also detect the existence of networks with non-broadcast SSIDs, and will capture, parse, and record data from such networks.

27. Kismet is a standalone application capable of capturing and filtering wireless frames. However, it can also be deployed in a configuration called a "drone," which does not record or analyze network traffic but instead forwards captured traffic to a server listening for such traffic. The Kismet drone program places a Kismet header describing the properties of the wireless transmission in front of the raw 802.11 frame and passes it to gslite for further processing. The gslite application listens for data from a Kismet drone running simultaneously within the Street View vehicle.

28. A Kismet drone is configured through the use of a file named kismet_drone.config, which provides, among other things, instructions for Kismet to "channel hop." Channel hopping is the act of cycling through numerous 802.11 channels per second in order to capture frames from as many nearby networks as possible. In the gstumbler project, Kismet's configuration file is created using a predefined template file, and entries in Google's template instruct the drone to change wireless channels five times per second, as shown below (kismet_drone.conf.template lines 37-41):

```
# Do we channelhop?
channelhop=true

# How many channels per second to we hop?  (1-10)
channelvelocity=5
```

As discussed above, the number of permitted channels for broadcast in a given frequency is regulated by a country's local authorities, and the number of permitted channels for broadcast in a frequency ranges between 11 and 14. The kismet_drone.conf.template file directs which channels should be monitored and the order through which they are hopped. In the United States, for example, there are 11 channels that may be used to wirelessly transmit data within the 2.4 Ghz band. Accordingly, when configured for the United States, Kismet listens to each of the 11 channels for one fifth of a second, thus listening to every channel for one 0.2 second interval during each 2.2 second channel hopping cycle.

III. Scope of Review and Methodology

29. Upon receipt of the gslite source code, Stroz Friedberg conducted a high-level review of the gslite framework code and associated modules. The purpose was to understand the basic logic flow and functionality of the program, and the significance and dependencies of the various components.

30. Based on our high level review, Stroz Friedberg identified key modules and dependencies for closer scrutiny, and assessed the significance of Google commands and code modules called from libraries external to the gslite code for use within the program. We received confirmation that particular functions and modules were borrowed from standard, shared libraries within Google. Because we also confirmed that such functions and codes were not customized for use in gslite, but were merely imported to perform standard functions, we focused on the core functionality and key programming modules unique to gslite.

31. We also did not independently review the Kismet program. As noted above, 802.11 frames initially are captured by the Kismet program, an open source packet sniffing program. It is our understanding based upon representations from Google that Kismet source code was not modified or adapted in any way as part of the gstumbler project.

32. We compared 802.11 frame specifications to the gslite frame parsing parameters encoded into the program to verify that the code's parameters are consistent with the specifications. That is, if the code parses particular bits of frame header information to determine, for example, the type of frame or whether the wireless network is encrypted, we confirmed that the program looks at the correct frame bits to parse the expected field from the raw data.

33. We closely scrutinized the parsing functionality of the gslite program as it pertains to each type of 802.11 frame. We determined how different types of frames are parsed, the different fields parsed for each frame type, what 802.11 frame fields are written to disk in parsed formats versus raw formats, and what 802.11 fields are discarded and not written to disk.

34. We analyzed the overall structure of code to determine the program's default behavior and the ways in which default behavior may be changed by command line arguments. We also examined the command line configuration settings over the course of gslite's deployment.

35. We confirmed our understanding as to other secondary functions of the program, including its logic to detect bad frames and not process them, its diagnostic capabilities for assessing proper functioning of the program, its calculation and correlation of GPS geolocation information with detected wireless networks, and its decision as to how and when to write data to disk.

36. Stroz Friedberg did not receive or analyze earlier versions of the gslite source code or its predecessors. We did, however, review the modification history and did not observe significant changes to the program regarding how frames are parsed and recorded. We also reviewed all available versions of the shell scripts used to launch Kismet and gslite to verify what command line arguments were used.

IV. Detailed Analysis and Findings

A. Source Code Flow and Functionality

37. At the highest level of description, Google's gstumbler program creates a series of servers and objects that interface with the Google Street View vehicle's GPS system and the Kismet drone, pulls wireless frames from a stream provided by the Kismet drone, and then assigns timestamp and geolocation information to each wireless frame it encounters, saving the results to disk. The general description of how gstumbler operates is illustrated in Figure 3, below, and in the following paragraphs.

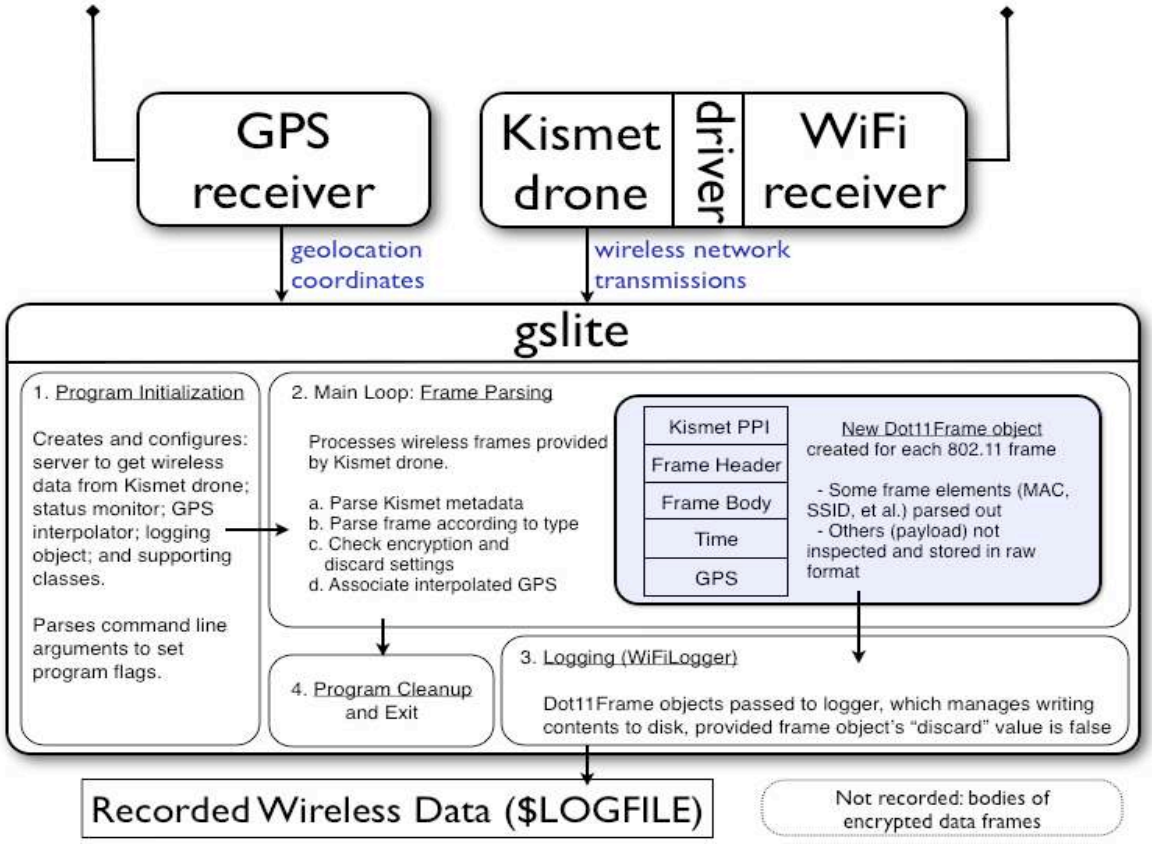


Figure 3: High-level representation of gslite program execution

38. The program first parses any command line arguments passed to it from the shell script, run_gstumbler, used to launch gslite. The program starts and configures a series of services, including, but not limited to: a WifiRecordLogger, which manages the storing of 802.11 frame data to disk; and a WifiLiteServer object, which listens for Kismet data on a predefined port.

39. For each frame being processed, the program creates a new Dot11Frame object in which to store the parsed 802.11 frame fields, along with a pointer to it. The Dot11Frame is a data structure that is built using Google's Protocol Buffers libraries. As noted previously, information about Dot11Frame objects and Protocol Buffers in general is provided in Appendix C.

40. The program parses the per-packet information (PPI) header information Kismet affixes to a captured 802.11 frame. PPI includes the quality of the signal, the signal strength, the signal noise, if the capture source indicated there was an error in the capture to Kismet, transmission channel, the signal carrier, the signal encoding, and the data transmission rate. The program

also sets the Dot11Frame's time received, time sent, and raw data properties to match those of the corresponding incoming frame.

41. The program proceeds to parse the 802.11 frame as described more fully in section B, below. The gslite program runs the Parse() method of a number of PacketParser objects against the incoming 802.11 frames: Dot11ParserImpl::Parse(); CtrlParerImpl::Parse(); MgmtParserImpl::Parse(); and TruncateParserImpl::Parse(). Although the forms of information available in a given frame vary according to its type and subtype, the packet parsers are applied to all frames regardless of type. The parsing process populates numerous properties of the Dot11Frame object with information extracted from the 802.11 frame. Parsing does not include inspection of the bodies of Data frames.

42. During the TruncateParserImpl::Parse() parsing function, gslite reads the encryption flag on each frame. That bit is located within the second byte of the Frame Control on an 802.11 frame. If the encryption flag is set to "true," then the frame's body, or payload, is cleared from memory and permanently discarded. If it is "false" the frame's body is retained for writing to disk.

43. The GPS interpolator associates geolocation coordinates with the frame and writes the coordinates into the Position property of the Dot11Frame.

44. The parsed 802.11 frame object is written to disk using WriteProtocolMessage() method of the RecordWriter object. In the case of Management frames, the body is written to disk as parsed Information Elements, while in the case of unencrypted Data frames, the body is written to disk in unparsed format. It is our understanding based upon representations from Google that the RecordIO module, used to write the Dot11Frame objects to disk, is a common shared library within Google, and it is utilized unchanged in gslite.

45. The main loop of the program continues parsing, collecting, and geolocating each 802.11 frame as it is detected and forwarded by the Kismet drone. An interrupt signal sent from a user or from the operating system will cause the program to exit the main loop, clean up objects in memory, and exit.

46. The gslite program also writes logging information, largely regarding program status and error conditions, to a default system location. Our review found one line of code that, when executed, writes the content of a wireless frame to disk, through the use of a protocol buffer method for formatting a data structure as a string (scanner.cc lines 114-115):

```
if (!parser_->Parse(frm)) {
      LOG(ERROR) << "Error parsing frame: " << frm->ShortDebugString();
```

The second line of code above writes the wireless frame to disk, including its body, regardless of frame type or encryption flag. However, the program only performs this logging when a wireless frame cannot be successfully parsed and the Parse() method returns false. Our review of the Parse() method determined that this condition is met only when a frame's length is too short to constitute a valid frame header. In such an event, the frame also would be too short to contain a frame body. Furthermore, any such invalid frame would be discarded by Kismet or the wireless card prior to being forwarded to gslite. Accordingly, the circumstances necessary to invoke this logging action preclude the possibility that frame payload content would be written to the error log.

47. During execution, gslite also reports certain diagnostic information in HTML format to the HTTP server to provide in-vehicle feedback regarding the status and operating state of gslite. This status monitor does not write output to disk.

48. Finally, we note that the gslite source code contains functions and methods that are never executed, and which appear to constitute vestigial or uncalled code. Stroz Friedberg

inspected such code but found no control flow that would lead to the execution of such code areas.

B. Frame Parsing

49. Following capture of the data by Kismet, gslite uses a Dot11Frame object to represent the structure of an 802.11 frame in memory, prior to writing the frame to disk. The gslite program processes these Kismet packets by removing the Kismet header, and then processing the underlying raw data, which is an 802.11 frame.

50. "Parsing" a property of an 802.11 frame results in its value being assigned to a property of Dot11Frame object, making it readily accessible for further analysis by gslite without additional decoding. Some 802.11 frame fields are analyzed by gslite and never assigned to a specific property of the Dot11Frame field object. Only some 802.11 frame fields are assigned to properties of Dot11Frame objects in their parsed form by gslite prior to being written to disk; others are stored in memory in a property field named “raw” and are written to disk without being further processed. By default, in the case of encrypted 802.11 Data frames, the frame’s body, which was temporarily stored in the Dot11Frame’s raw field, is cleared from memory and never written to disk.

51. Specifically, gslite parses all available 802.11 frame header information and stores those properties in memory in a Dot11MacHeader object. The remaining frame data, the body, is stored in its raw form in the raw property field of a Dot11FrameBody object. A Dot11MacHeader object is a representation of the 802.11 frame header in the memory of a computer. Similarly, a Dot11FrameBody is a representation of the body or payload of an 802.11 frame body.

52. The Dot11MacHeader’s properties and the Dot11FrameBody object may be further analyzed or parsed depending on the type of frame. Dot11FrameBody objects contain ManagementFrameBody and ControlFrameBody objects to represent metadata specific to Management and Control frames respectively:

a. Control frames undergo the least additional analysis as they contain comparatively less data than other frame types. Only the subtype information from an 802.11 Control frame’s Frame Control field will be parsed and stored in memory as its own parsed property.

b. Management frames, which contain the administrative information necessary to manage wireless transmissions, undergo both additional analysis, and parsing. Management frames’ Frame Control properties are analyzed to determine the values of the To DS and From DS fields, which indicate the number of MAC addresses within the frame; however, these values are not stored in their own property fields in memory. Furthermore, Management frames’ bodies are parsed and stored as a series of Information Elements in the ManagementFrameBody’s collection of InformationElement objects. Included in the Information Elements properties is the SSID. The gslite program parses and stores the SSID information for all wireless networks, whether the SSID is broadcast or not. Any extra data stored in the ManagementFrameBody is stored in the “extra” property. Once this process is complete, the raw property of the Dot11FrameBody object is then cleared for Management Frames.

53. Although Data frame header information is further analyzed during the parsing process, Data frame bodies are not parsed. Specifically, gslite analyzes a Data frame's Frame Control field to determine the values of the To DS and From DS fields contained therein; however, these values are not parsed or stored in their own properties in memory.

54. In summary, the parsing function of the gslite program does the following:

a. All 802.11 frames have all of their available 802.11 frame header information parsed and stored in properties of a Dot11MacHeader object in memory, regardless of frame type. A frame's body will be stored as raw data in a Dot11FrameBody's raw property, and this raw data may be further parsed if the frame is a Management Frame. The frame type information from a frame's Frame Control field is parsed and stored in memory as its own value, regardless of frame type.

b. If the frame is a Control frame, the subtype information from the Frame Control field will be parsed and stored in memory as its own value. No additional parsing is performed on Control frames.

c. If the frame is a Management frame, the To DS and From DS fields from the Frame Control field are analyzed, but are not parsed and stored in memory as their own properties. Management frame bodies are parsed and stored as a series of Information Elements in ManagementFrameBody's collection of InformationElement objects (which is in the Dot11Frame's Dot11FrameBody object). Any extra data in the body is stored in the ManagmentFrameBody's "extra" property, and the "raw" property of the Dot11FrameBody object is cleared.

d. If the frame is a Data frame, the To DS and From DS fields from the Frame Control field are analyzed, but are not parsed and stored in memory as their own properties. Data frame bodies are not parsed. As discussed more fully below, the body of a Data frame is discarded if the Protected Frame bit is set to "true", which indicates the frame is encrypted; otherwise, the body is written as unparsed data to disk.

C. Default Settings Governing Discard of Data and Writing to Disk

55. After gslite's program logic parses each 802.11 frame according to its type, a Dot11Frame object exists with all available frame properties parsed and stored in the appropriate property fields. At this point in the execution of the program, the program's settings are checked to determine whether or not to retain the current frame data in whole or in part.

56. By default, gslite records all wireless frame data, except for the bodies of Data frames from encrypted wireless networks. The code governing whether data elements of a frame should be retained or discarded occurs in the file "packetparserimpl.cc." Four variables, or flags, are assigned default Boolean values to establish the program's default behavior regarding what to discard from memory and what to retain. In particular, the default settings, as shown below, are set to discard the bodies of encrypted frames[3] and to retain everything else (packetparserpmpl.cc lines 14-21):

```
DEFINE_bool(discard_encrypted_body, true,
      "Discard bodies of encrypted 802.11 frames");
DEFINE_bool(discard_control_frame, false,
      "Discard 802.11 control frames");
DEFINE_bool(discard_data_frame, false,
      "Discard all 802.11 data frames");
DEFINE_bool(discard_management_frame, false,
      "Discard all 802.11 management frames");
```

[3] Although a Management frame of the subtype Authentication would have its encryption flag set to "true," the sequence of the execution path causes such Management frame bodies to be stored in the "extra" property and written to disk. Management frames do not contain user content.

57. The same file, packetparserimpl.cc, contains the code that checks each wireless frame processed and determines whether or not to retain it in whole or in part, based upon the Boolean values of the flags defined above. The program checks to see whether the "discard_encrypted_body" flag is set to "true", which is the default setting. If so, gslite checks the frame being parsed to see whether its encryption flag is set to "true." If both checks return "true" then the frame is encrypted and the program discards the encrypted frame's body. The frame body is cleared, using the accessor method clear_body().

```
if (FLAGS_discard_encrypted_body && PacketUtil::IsEncrypted(f)) {
      // Discard just the body of encrypted frames
      f->clear_body();
```

Subsequently, when the remainder of the frame is written to disk, its body is not recorded.

58. The program checks the type of the frame being parsed (that is, whether it is a Control, Data, or Management frame) and then checks the value of the corresponding Boolean flag from among the discard flags above. If it is "true", the discard flag of the current frame object is set using the Dot11Frame accessor method set_discard(true).

```
switch (PacketUtil::Type(f)) {
case Dot11FrameBody::CONTROL:
      if (FLAGS_discard_control_frame)
            f->set_discard(true);
      break;
case Dot11FrameBody::DATA:
      if (FLAGS_discard_data_frame)
            f->set_discard(true);
      break;
case Dot11FrameBody::MANAGEMENT:
      if (FLAGS_discard_management_frame)
            f->set_discard(true);
      break;
default:
      break;
}
```

59. At a subsequent point in program execution when a parsed frame is to be written to disk, the discard flag of the frame object is checked: if the flag is set to "true", the frame is not written to disk (scanner.cc lines 105-111):

```
void WifiScanner::TryLog(Dot11Frame * frm) {
  if (is_logging_ &&
      logger_ &&
      !frm->discard() &&
      !logger_->Write(frm))
    LOG(ERROR) << "Error writing to log";
}
```

D. GPS Interpolation

60. The onboard GPS system provides geolocation coordinates at some rate slower than the rate at which wireless frames can be received. Accordingly, gslite interpolates the position at which each wireless frame was received and associates the interpolated position with the frame object. Stroz Friedberg's review of source code relating to GPS coordinate interpolation found no code execution paths that would affect the wireless data written to disk by gslite.

E. Command Line Arguments in Configuration Files

61. The Boolean flag definitions set forth in section C above provide the default program behavior. However, the flags can be superseded by command line arguments defined in accordance with Google's coding standards. The first line of code executed by gslite processes any and all command line arguments (see gslite.cc lines 12 and 128-129, below). It is our understanding from Google that InitGoogle(), a method defined outside the scope of the provided source code, sets the values of program variables using the command line arguments. The Google standards for using command line flags is documented at http://google-gflags.googlecode.com/svn/trunk/doc/gflags.html.

```
#include "base/commandlineflags.h"
...
int main(int argc, char** argv) {
  InitGoogle(argv[0], &argc, &argv, true);
```

62. Command line arguments will supersede the default values for the discard and encryption flags discussed above and change the behavior of gslite. Since the flag "discard_data_frame" is false by default, gslite will discard entire Data frames if and only if the flag "discard_data_frame" is run on the command line at the time of program execution (or until such time as the default behavior is revised in source code).

V. Conclusion

63. Gslite is an executable program that captures, parses, and writes to disk 802.11 wireless frame data. In particular, it parses all frame header data and associates it with its GPS coordinates for easy storage and use in mapping network locations. The program does not analyze or parse the body of Data frames, which contain user content. The data in the Data frame body passes through memory and is written to disk in unparsed format if the frame is sent over an unencrypted wireless network, and is discarded if the frame is sent over an encrypted network.

APPENDIX A

INVENTORY OF REVIEWED SOURCE CODE FILES AND SHELL SCRIPTS

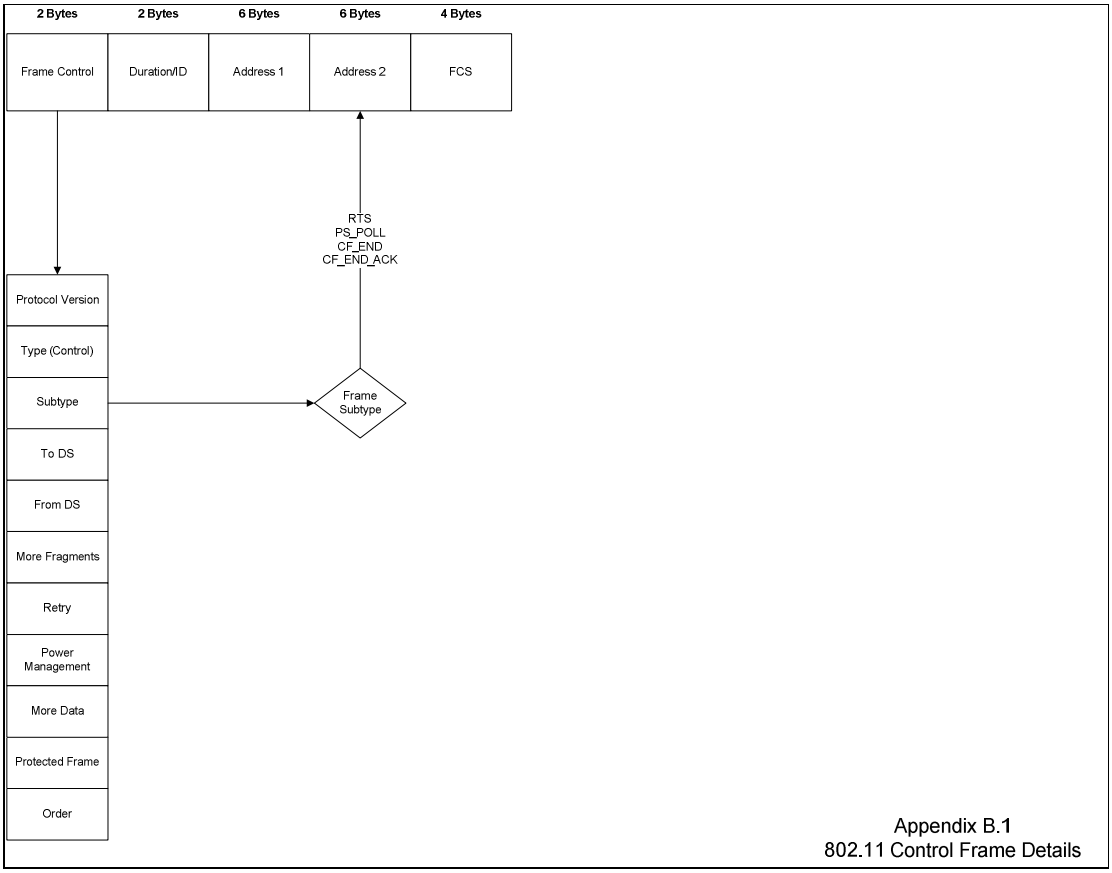
Stroz Friedberg reviewed the following provided C++ source code, configuration files, and shell scripts as part of its static source code analysis. The dates of last modification are derived from the compressed tar files in which the source code was provided and are believed to correspond to the dates of modification of official, checked-in source code.

File Name	Last Written On	SHA-1 Hash Value
gstumbler Source Code Provided as gstumbler-src.tgz on 5/20/2010		
BUILD	7/1/2009	7de19d35307cfdc9fc8c03c9d8d44aee3cebcbaa
gps_messages.h	3/31/2010	aa9cef443f3e1352056751cdc3ca8d35705cbf1f
gps-interpolator.cc	11/7/2007	37001680b7e4acd0410fd890523fa911371cdf63
gps-interpolator.h	4/30/2008	688d310771e66e2ecc92c7069059bda2e378d1d8
gps-interpolator_test.cc	2/2/2010	21e241b6cdb0ae65f2d395f38d5541d0ef2b3ed8
gps-ipc.cc	3/31/2010	2413c0538add232332fa25ba1498274f54e2d76f
gps-ipc.h	3/31/2010	175193adb5116594e6f644c9b9bb8a9920476d8a
gps-ipc_test.cc	3/31/2010	3ea76455f6fd12391c6e60ad9d8b0fe9bffb0db4
gslite.cc	3/31/2010	796c67b420ffd5ff0afbc65c42c07d08256686d3
gstumbler.cc	4/30/2008	2104989fdc44b9c53acbf5bc6857ee8f1fc2594e
gstumbler-run.sh	3/5/2007	e5045fac3b9e6de3ce36b3b797e504a9c741254a
kismetconnection.cc	6/19/2009	4b3cb2dcfef03c53bdf3f46088039c1105d29fe3
kismetconnection.h	6/19/2009	cacb6ca54136cc1bcf3a64f9a54a25b4939f2a7f
logger.cc	11/7/2007	03f2733398191d36fae6297564b455086bdfda83
logger.h	11/7/2007	83df2ff3e50f5e070af8f4acf1c032ca6a2f8682
monitor.cc	10/31/2006	7b5381eb9adeb12e09589f84e817f170bc783ade
monitor.h	10/31/2006	64870c0f3df0b169ef352b0c3f920bd48ff6073c
packet.proto	3/31/2010	872e43bb2477b3d50dfdd34f68adad7290f49f6c
packetparser.cc	10/31/2006	f42687c8f5bef580ce46476eb840e0022280d969
packetparser.h	7/1/2009	3855b17808778d752824ea6a2efbe875307933ac
packetparser_test.cc	2/2/2010	dc795a3e99ec890db87d1e97ac835ed3f74a3f7b
packetparserimpl.cc	10/31/2006	ec094b96ab14ba7bf251160ad6d3285d4fa3a714
packetparserimpl.h	10/31/2006	d8f5c40b3954133c8be46e6cabf9f23f91de6ecc
packetsource.cc	10/31/2006	bfe6dec9aa9d4a4095c0ad34c9f103b7344154d5
packetsource.h	3/4/2010	69f2b4ffa32e925e56bdf0f56097cf5bd7ce0ed9
packetsourceimpl.cc	12/16/2009	75828b368c1682ebac547c1193e9d3fbcc27f54a
packetsourceimpl.h	7/1/2009	bff09f7f55cdd080eaf1d9057a8a33c1d9cbb8f8
packetutil.h	1/28/2008	8dedee1c5b43811bd7a16ea9b5afc58b69adf2f2
resources\drive_status.tpl	10/18/2007	065c489ee01d5de2ff85f92829fceeebd58359e9
scanner.cc	3/31/2010	33d4a92a87a679faf0932e492ffbe6cf32a9534a
scanner.h	3/31/2010	4a869a3f54a4f2662c09b8fd90e4e14bf631cb83
scanner_test.cc	2/2/2010	7a8004d0c19cc1337ca9cb888bd3f7830a26413b
Configuration files and shell scripts -- most recent versions Provided as gstumbler-config.tgz on 5/20/2010		
config_interfaces.sh	5/18/2010	51c00340e9744dda850ca0ee546bcce067327caa
kismet_drone.conf.template	5/18/2010	f5bd93b3fc1ba8ada0827cc04fc6ca5c24aab99c

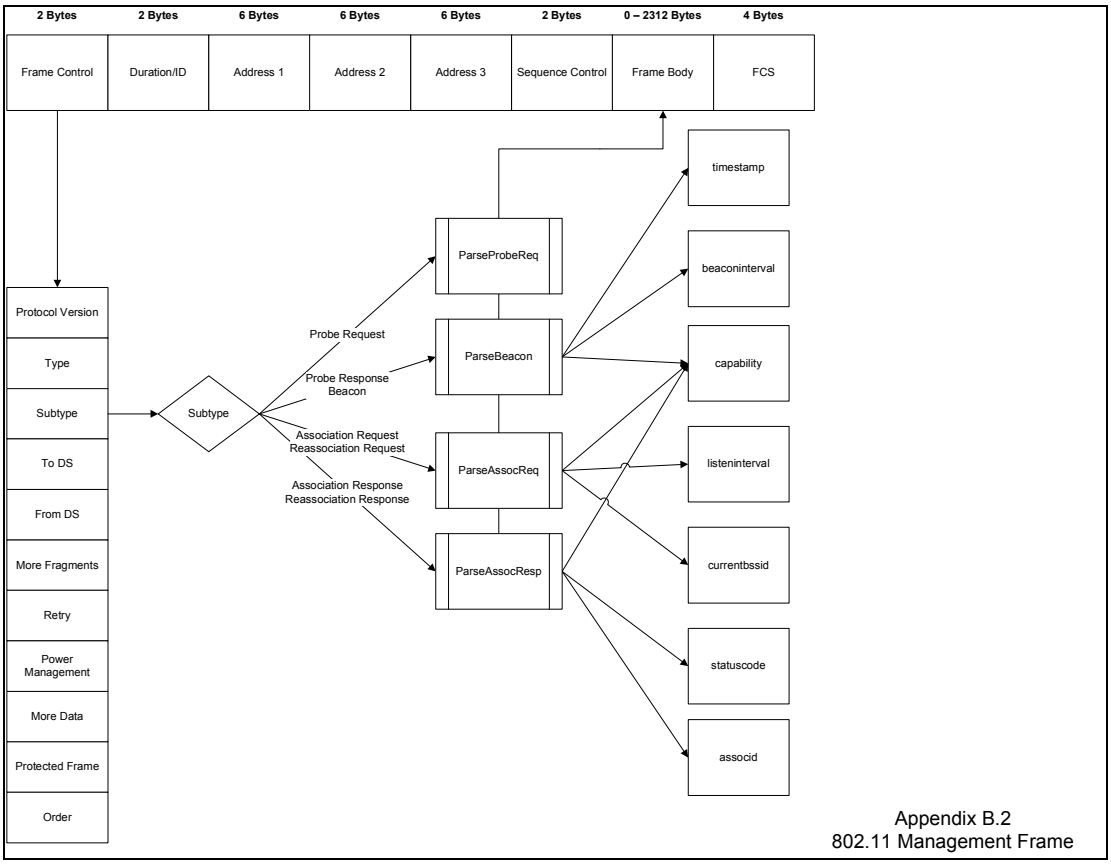
run_gstumbler.template	5/18/2010	7b3aacb15f8b878b8bd91d34242c6b4a1e958691
run_kismet	5/18/2010	7c8b2b13061b6cb8280256556910d56b93848a20
Configuration files and shell scripts -- historical versions Provided as gstumbler-scripts2.tgz on 5/26/2010		
config_interfaces.sh#1	5/26/2010	7b85ea7c7babd7a7f15f0caa1fc1e3a2814f9d75
config_interfaces.sh#2	5/26/2010	faeeebfae425597af82acebdedccc2c972088b10
config_interfaces.sh#3	5/26/2010	5816de44b2cf67116958e7bd35240bf1f3186953
config_interfaces.sh#4	5/26/2010	fc5ee14d002970d532ec55cee09962959b78d28b
run_gstumbler.template#1	5/26/2010	9a718b8727a2c590e670fc08ea27fa4818309253
run_gstumbler.template#2	5/26/2010	4f4ca3f5d2175eecadf1c104a8aba702cce34778
run_kismet#1	5/26/2010	27df00844852cd7e0070d82324ab5cc2fb81881c
Supporting library for managing record writing Provided as bulkstorage.tgz on 5/26/2010		
bulkstorageblock.h	11/1/2006	d7240f808766bd718e80f1293dcaba95ff50af18
bulkstoragewriter.cc	3/12/2007	e361e6c9d16cc64af15bb3df6a6cfdd58e049b6f
bulkstoragewriter.h	3/12/2007	d0dad037253f4f83a9107c7ea004c8d8e26f78d1
bulkstoragewritermanaged.cc	3/4/2010	bab20ee94c25d62c2d8a18259915bf0906d68115
bulkstoragewritermanaged.h	3/4/2010	1d8b67f468f0b3d7dbe4f609548261b37fed4eb0
disk_write_methods.cc	3/12/2007	134aea15d93f667e322e7c70c7b89609755e2052
disk_write_methods.h	12/29/2006	4609dcf39b55cc2e111f338b7dbc4a3caf891109
performancemonitor.cc	8/10/2007	f4aece5bd4bcbd520e654ab0d9802c560c2efc09
performancemonitor.h	11/29/2006	b8c37eb8a427fdd72f707985661a71641c7436ec
sectensecminstats.cc	11/29/2006	34d884b123216a4fb5bd640bf51d2e8f2ad42ef1
sectensecminstats.h	6/22/2009	38c8bf84879ecdade44a31642b5aba0e30e6cccd

APPENDIX B

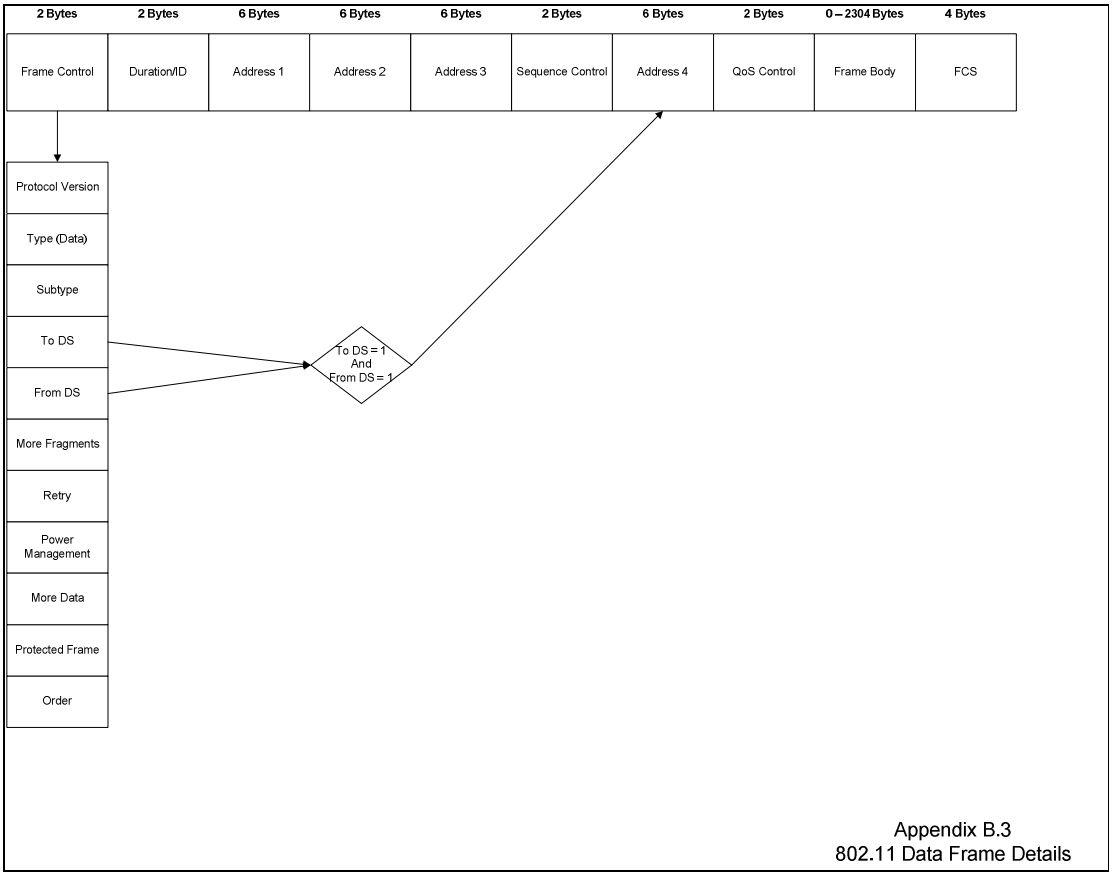
802.11 FRAME ELEMENTS



Appendix B.1
802.11 Control Frame Details



Appendix B.2
802.11 Management Frame


2 Bytes
2 Bytes
6 Bytes
6 Bytes
6 Bytes
2 Bytes
6 Bytes
2 Bytes
0 – 2304 Bytes
4 Bytes
Frame Control
Duration/ID
Address 1
Address 2
Address 3
Sequence Control
Address 4
QoS Control
Frame Body
FCS
Protocol Version
Type (Data)
Subtype
To DS
From DS
More Fragments
Retry
Power Management
More Data
Protected Frame
Order
To DS = 1 And From DS = 1
Appendix B.3
802.11 Data Frame Details

APPENDIX C

THE GSTUMBLER DOT11FRAME PROTOCOL BUFFER AND SUMMARY OF RECORDED CONTENT

C-1. Google source code employs a serialization format, accomplished through the use of objects developed at Google called Protocol Buffers, which are used to exchange and write structured data. Protocol Buffers take an object representing a complex data structure and transform that structured object into a bitstream, suitable for transmission or writing to disk, through a transformation called serialization. The source code for protocol buffers was released under an open source license by Google in 2008. An overview of documentation regarding protocol buffers is available at (http://code.google.com/apis/protocolbuffers/docs/overview.html).

C-2. Each type of object to be serialized is specified as a Protocol Buffer "message," which establishes the structure of each object type. In the gstumbler project source code, Protocol Buffers are declared in the file packet.proto. The protocol buffer message of central importance to gslite's functionality is the Dot11Frame object, a message that is a structured representation of a single 802.11 wireless frame. The Dot11Frame object contains multiple other protocol buffer messages, also defined in packet.proto, that represent various components and types of wireless frames.

C-3. Protocol buffers provide accessor functions to set and retrieve the values of fielded data within a message. Standard accessor functions include get_<fieldname>, set_<fieldname>, and clear_<fieldname>, where <fieldname> is one of the defined data elements within the message. As discussed in paragraphs 57 and 58 of this report, the Dot11Frame accessor methods clear_body() and set_discard(true) will be called if certain flags and conditions are true. These methods serve, respectively, to clear only the content of the Dot11Frame's Body field and to set the Discard Boolean flag of a Dot11Frame message to true. These two methods are the means by which a frame is written to disk without its payload or not at all.

C-4. The following tables summarize the properties within each of the protocol buffer messages defined in packet.proto.

Dot11Frame Object	
Property	**Description**
Raw	A buffer used to store the unprocessed data; this buffer contains the raw frame data parsed throughout frame processing and is cleared prior to the data being written to disk.
Header	A Dot11MacHeader object in the protocol buffer message format described below.
Body	A Dot11FrameBody object in the protocol buffer message format described below.
Position	A cityblock.PositionInfo object containing GPS coordinates.
PositionComment	An optional string.
TimeRecvd	The time the frame arrived for processing.
TimeSent	The estimated time the frame was transmitted.
KisMetadata	A KismetMetadata object, described below, containing per-packet information including 802.11 channel, signal quality, and frame length.
Discard	A boolean flag that indicates whether or not the entire frame – metadata and body – should be written to disk.

Dot11MacHeader	
Property	**Description**
Raw	The raw data buffer containing the data that is processed and stored in the header's fields.
FrameControl	A thirty-two bit integer used to store the sixteen bit Frame Control field in an 802.11 frame.
DurationOrId	A thirty-two bit integer used to store the sixteen bit field in position bytes 2 to 3 in an 802.11 frame. These sixteen bits are either the duration or id depending on the type and subtype of the frame.
Address1	The first Media Access Control (MAC) address in an 802.11 frame. A MAC address is a six byte hexadecimal address specifying a network device.
Address2	The second MAC address in an 802.11 frame.
Address3	The third MAC address in an 802.11 frame.
SequenceControl	The sixteen bit sequence control number present in data and management frames. Data may be fragmented for transmission or re-transmission. If the data is fragmented, this number is used to determine where in sequence a fragment fits. This field is zero for the first or only fragment of data, and incremented for each successive fragment sent.
Address4	The fourth MAC address in an 802.11 frame.
QoSControl	Sixteen bits of quality of service related information and policies sent by hardware supporting quality of service.

Dot11FrameBody	
Property	**Description**
Raw	The raw data buffer containing the data that is processed and stored in the body's fields.
FrameType	An enumerated type that specifies if a frame is: a Management frame (0); a Control frame (1); a Data frame (2); a Reserved type frame (3); or if there is no frame type detected (9999).
Ctrl	An optional ControlFrameBody object, defined below.
Mgmt	An optional ManagementFrameBody object, defined below.

ControlFrameBody	
Property	**Description**
Subtype	An enumerated type specifying the subtype of a Control frame. Its potential values are: PS_POLL (10); RTS (11); CTS (12); ACK (13); CF_END (14); CF_END_ACK (15); and NO_CTRL_SUBTYPE (9999).

ManagementFrameBody	
Property	**Description**
Subtype	An enumerated type specifying the subtype of a Management frame. Its potential values are: ASSOC_REQ (0); ASSOC_RESP (1); REASSOC_REQ (2); REASSOC_RESP (3); PROBE_REQ (4); PROBE_RESP (5); BEACON (8); ATIM (9); DISASSOC (10); AUTH (11); DEAUTH (12); and NO_MGMT_SUBTYPE (9999).
AuthAlgorithm	A thirty-two bit integer that is not set in the code reviewed.
AuthTransaction	A thirty-two bit integer that is not set in the code reviewed.
BeaconInterval	A thirty-two bit integer that is used to store the sixteen bit value of the number of time units between target beacon transmission times.
Capability	A thirty-two bit integer that is used to store the sixteen bit series of flags outlining the functionality of the transmitter.

CurrentBSSID	A sixty-four bit integer that is used to store the fourty-eight bit MAC address of the access point with which the transmitter is currently associated with.
ListenInterval	A thirty-two bit integer used to store the sixteen bit value of how often a receiver in power saver mode wakes to listen to Beacon mangement frames.
ReasonCode	A thirty-two bit integer that is not set in the code reviewed.
AssocID	A thirty-two bit integer that is used to store the sixteen bit value assigned by an access point during the association process.
StatusCode	A thirty-two bit integer that is used to store the value used in a response management frame to indicate the success or failure of a requested operation.
Timestamp	A sixty-four bit integer used to store the value of the timing synchronization function timer of a frame's source.
IEs	A collection of Information Elements, or key-value pairs regarding a transmitter.
SSID	A string containing the name of the access point.
Channel	A thirty-two bit integer used to store the channel on which a frame was sent.

KismetMetadata	
Property	**Description**
hdrlen	A thirty-two bit integer used to store the length of the Kismet header.
drone_ver	A thirty-two bit integer used to store the sixteen bit value of the version of the Kismet drone.
datalen	A thirty-two bit integer used to store the length of the data captured by Kismet.
caplen	A thirty-two bit integer used to store the length of the data originally captured by Kismet.
tv_sec	A sixty-four bit integer storing a timestamp in seconds.
tv_use	A sixty-four bit integer storing a timestamp in microseconds.
quality	A thirty-two bit integer used to store the sixteen bit value signal quality.
signal	A thirty-two bit integer used to store the sixteen bit value signal strength.
noise	A thirty-two bit integer used to store the sixteen bit value signal noise level.
error	A thirty-two bit integer used to store the eight bit value whether the capture source told Kismet the frame was bad.
channel	A thirty-two bit integer used to store the eight bit value of the hardware channel that received the frame.
carrier	A thirty-two bit integer used to store the eight bit value of the signal carrier.
encoding	A thirty-two bit integer used to store the eight bit value of the signal encoding.
datarate	A thirty-two bit integer used to store the value of the data rate, which is in units of 100 kbps.
adapter	A thirty-two bit integer used to store the mapped value of an adapter name.